Document made available under the Patent Cooperation Treaty (PCT)

International application number: PCT/EP05/050028

International filing date: 05 January 2005 (05.01.2005)

Document type: Certified copy of priority document

Document details: Country/Office: US
Number: 60/534,294
Filing date: 05 January 2004 (05.01.2004)

Date of receipt at the International Bureau: 09 May 2005 (09.05.2005)

Remark: Priority document submitted or transmitted to the International Bureau in compliance with Rule 17.1(a) or (b)

World Intellectual Property Organization (WIPO) - Geneva, Switzerland
Organisation Mondiale de la Propriété Intellectuelle (OMPI) - Genève, Suisse

PA 1310256

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

April 21, 2005

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE UNDER 35 USC 111.

APPLICATION NUMBER: *60/534,294*
FILING DATE: *January 05, 2004*

EP/05/50028

By Authority of the
COMMISSIONER OF PATENTS AND TRADEMARKS

M.K. Hawkins

M. K. HAWKINS
Certifying Officer

PTO/SB/16 (08-03)

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c).

Express Mail Label No.	EV 338468330 US

INVENTOR(S)		
Given Name (first and middle [if any])	Family Name or Surname	Residence (City and either State or Foreign Country)
Filip	Verhaeghe	Brussels, Belgium
Stefaan	Decorte	Brussels, Belgium

☐ *Additional inventors are being named on the separately numbered sheets attached hereto*

TITLE OF THE INVENTION (500 characters max)

BEHAVIOR-BASED MULTI-AGENT SYSTEMS (BB-MAS) AS DATA TYPES

Direct all correspondence to: **CORRESPONDENCE ADDRESS**

☒ Customer Number **20350**

OR

☐ Firm *or* Individual Name					
Address					
Address					
City		State		ZIP	
Country		Telephone		Fax	

ENCLOSED APPLICATION PARTS *(check all that apply)*

☒ Specification *Number of Pages* 30 ☐ CD(s), Number

☒ Drawing(s) *Number of Sheets* 1 ☒ Other (specify) 27 Pages Appendix

☒ Application Data Sheet. See 37 CFR 1.76

METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT

☒ Applicant claims small entity status. See 37 CFR 1.27.

☐ A check or money order is enclosed to cover the filing fees

FILING FEE AMOUNT ($)

☒ The Director is hereby authorized to charge filing fees or credit any overpayment to Deposit Account Number: **20-1430** — 80

☐ Payment by credit card. Form PTO-2038 is attached.

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

☒ No.

☐ Yes, the name of the U.S. Government agency and the Government contract number are: .

Respectfully submitted,

SIGNATURE Gerald T. Gray

TYPED or PRINTED NAME Gerald T. Gray

TELEPHONE 925-472-5000

Date 01/05/04

REGISTRATION NO. 41,797 (*if appropriate*)

Docket Number: 022157-000100US

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

60111403 v1

Application Data Sheet

Application Information

Application number::	
Filing Date::	01/05/04
Application Type::	Provisional
Subject Matter::	Utility
Suggested classification::	
Suggested Group Art Unit::	
CD-ROM or CD-R??::	
Number of CD disks::	
Number of copies of CDs::	
Sequence Submission::	
Computer Readable Form (CRF)?::	
Number of copies of CRF::	
Title::	BEHAVIOR-BASED MULTI-AGENT SYSTEMS (bb-MAS) AS DATA TYPES
Attorney Docket Number::	022157-000100US
Request for Early Publication::	No
Request for Non-Publication::	No
Suggested Drawing Figure::	1
Total Drawing Sheets::	1
Small Entity?::	Yes
Latin name::	
Variety denomination name::	
Petition included?::	No
Petition Type::	
Licensed US Govt. Agency::	
Contract or Grant Numbers One::	
Secrecy Order in Parent Appl.::	No

Applicant Information

Applicant Authority Type::	Inventor
Primary Citizenship Country::	Belgium
Status::	Full Capacity
Given Name::	Filip
Middle Name::	
Family Name::	Verhaeghe
Name Suffix::	
City of Residence::	Brussels
State or Province of Residence::	
Country of Residence::	Belgium
Street of Mailing Address::	
City of Mailing Address::	Brussels
State or Province of mailing address::	
Country of mailing address::	Belgium
Postal or Zip Code of mailing address::	

Applicant Authority Type::	Inventor
Primary Citizenship Country::	Belgium
Status::	Full Capacity
Given Name::	Stefaan
Middle Name::	
Family Name::	Decorte
Name Suffix::	
City of Residence::	Brussels
State or Province of Residence::	
Country of Residence::	Belgium
Street of Mailing Address::	
City of Mailing Address::	Brussels
State or Province of mailing address::	
Country of mailing address::	Belgium
Postal or Zip Code of mailing address::	

Correspondence Information

Correspondence Customer Number:: 20350

Representative Information

Representative Customer Number:: 20350

Domestic Priority Information

Application:: Continuity Type:: Parent Application:: Parent Filing Date::

Foreign Priority Information

Country:: Application number:: Filing Date::

Assignee Information

Assignee Name::
Street of mailing address::
City of mailing address::
State or Province of mailing address::
Country of mailing address::
Postal or Zip Code of mailing address::

Attorney Docket No.: 022157-000100US

PROVISIONAL

PATENT APPLICATION

BEHAVIOR-BASED MULTI-AGENT SYSTEMS (BB-MAS) AS DATA TYPES

Inventor(s): Filip Verhaeghe, a citizen of Belgium, residing at

Brussels, Belgium

Stefaan Decorte, a citizen of Belgium, residing at

Brussels, Belgium

Assignee: Agent Business Force
Avenue Joseph Wybran 40,
Brussels, 1070
Belgium

Entity: Small business concern

TOWNSEND and TOWNSEND and CREW LLP
Two Embarcadero Center, Eighth Floor
San Francisco, California 94111-3834
Tel: 925-472-5000

Behavior-Based Multi-Agent Systems (bb-MAS) as Data Types

Inventors: Filip Verhaeghe, Stefaan Decorte; for Agent Business Force NV.

The present invention provides novel agent-based programming systems and methods as well as systems and methods for implementing agent systems. Specific programming language concepts are described herein with reference to a specific language implementation, termed RIDL (Robot Intelligence Definition Language). It should be appreciated, however, that other implementations using the various concepts and aspects as described herein may be implemented without departing from the invention. Appendix A, which discloses one specific implementation, RIDL2004, is presented as an integral part of this document.

RIDL is a programming language that, in one aspect, adds several keywords over traditional languages. RIDL as a superset of Java and as a superset of C# have been implemented, however, it is understood that other language constructs may be used besides C# and Java . Hence, RIDL as described herein includes an abstract description of the various concepts, principles and methods that drive the RIDL application family.

There are other entities that are creating multi-agent development systems by offering libraries that can be used from object-oriented languages. What is presented herein represents a leap forward relative to these systems. No longer do you need to learn libraries, as the language natively supports agents. But the key is that this allows us to present a totally new concept: *Agents as data types*. It provides a unifying view on what agents are, and what objects are. It provides new levels of ease of use, which in turn allow more complex systems to be built.

Another way of properly understanding the various aspects of the present invention is the following explanation: unlike every single competing system, we do not wish to reduce agents to objects, such that they fit into the object-oriented framework. Rather, we wish to lift objects to the agent-oriented framework.

- For the whole world, agents are a kind of objects with added functionality.
- For us, objects are a kind of agents who are missing functionality.

Understanding this paradigm shift is fundamental to understanding aspects of the present invention. It represents a shift in the way one views the world, and with that shift comes new ease of use and new insights. This paradigm shift is at the core of the technicalities of the present invention.

As will be described herein, the language, methodology and implementation of RIDL will be described with reference to a compiler, however, it is understood that the invention is not limited to compilation, and the inventin is also not limited to the particular syntax and implementations as described herein. For example, aspects of the invention may be implemented using an interpreter or other language reading engine.

COMPILER LEVEL ASPECTS

Agents as data types

The added keywords make the creation of multi-agent systems very straightforward. A keyword "agent" is added at the same level as a traditional "object". The keywords agent and object are interchangeable from syntactical point of view. The effect of the keyword indicates that the block defines an agent. Notice that other words than "agent" could be used. It would also be possible to annotate the object with attributes that indicate that this object is in fact an agent. Whatever the syntax, the objective is to indicate that what is enclosed is an agent, and behaves as such.

According to one aspect of the present invention, agents are considered as a native data type, exactly like objects. All actions that can be done with objects, can also be done with agents. Examples of such actions are creation of new agents, destruction of agents, passing agents as parameters, creating references (pointers) to agents, duplicating agents, etc.

Like with objects, agent definitions actually define classes of agents. Somewhere in the code, an agent *instance* is created by a *new* statement, or when a new variable of that type (an agent class) is defined.

Ultimately, a language could be defined in which all objects are by definition agents, and hence one needs not mark objects as being agents. Simply by looking if the features decribed below are used (sensors and/or behaviors), the compiler could distinguish between active agents (agents) and passive agents (objects). Therefore, agents as a data type can only be completely understood by taking the features described below into account.

Sensors

A *sensor* is a variable of traditional type that is annotated as a sensor. This annotation can be done by adding an attribute to the variable, or by using a new keyword to indicate that a special variable is defined. In the code sample, the attribute "sensor" is used to annotate an existing variable.

A sensor is a part of a multi-agent system. It is almost always part of a specific agent, but a sensor could also be defined as "global", in which it isn't part of any particular agent but available to all agents (and hence it cannot "die" as agents can).

When a sensor is updated or changed, it throws an *event* to indicate what has happened. *Sensor events* could for instance be *updates*, *changes*, etc. Anyone can listen to these events, as we will see further.

At this time, the language C# already has an annotation that allows a variable to throw events when something happens to the variable. However, to receive this event, the listener must *register* onto the throwing variable. According to one aspect of the present invention, the listener does *not* need to register to receive the event. Behaviors can listen for types of events, and any event that fits the type will be picked up by the behaviors. This innovation is important. In multi-agent systems, the number of agents is dynamic. A behavior *does not necessarily know* which agents are present. Hence, it cannot register onto these agents. This functionality will be revisited when the *any* operators are discussed.

To use a descriptive metaphor: suppose you are having a telephone conversation. In C#, you can talk to anyone you know is at the other end of line. However, if somebody on the other end says something and you didn't know he was going to be present, you don't

hear him. In RIDL, you will hear everyone that is speaking on the other side, and you can converse with all of them, even if you didn't expect them to be there or if you have never heard of them.

This innovation is an important aspect of dealing with uncertainty of the environment, and with continuous changing populations of agents and changing environments.

In C# and related programming languages, the designer explicitly needs to throw an event. The designer himself has to write code to call a method on the event, in order to notify listeners (e.g. "thowEvent(myEvent)"). To accommodate for the uncertainty of the environment, RIDL presents another implicit innovative aspect:

- In all existing languages, the thrower decides to throw the event;
- In RIDL, the listener/catcher decides whether the event is recorded.

Behaviors

A *behaviors* is a *method* (a procedure that is part of an object, or in our case, that is part of an agent) that is annotated as a behavior. This annotation can be done by adding an *attribute* to a method, or by using a *new keyword* to indicate that a special method is defined. In the code sample, a new keyword "*behavior*" is used to indicate such a special method.

Behaviors have their own events. Every time a behavior is started, it throws an *activates* event, and upon completion, it throws a *completes* event. Other events may be defined, and the names of the events indicated here are for clarification only, as the exact syntax may vary. Examples of other potential events are *suspended*, *born*, *died*, *waiting*, etc. Any number of events can be implemented. One important innovative aspect is that these events provide information to other behaviors and other agents about its status, and that this functionality is *implicit*. A behavior will throw these events without any action on the part of the developer.

Behaviors are very different from traditional methods in the way they are activated. Traditional methods, as they are defined in object-oriented methodologies, are always invoked by some other method. A method may potentially be invoked as part of the creation of a new thread of execution, but even in this case, it is some external logic that determines when the method is called. By contrast, behaviors are completely in control over when they are activated. Behaviors cannot be invoked by external logic, only they themselves decide when to activate. Behaviors do not need external code to suddenly become active.

A behavior has a section that indicates when it is should be activated. This section contains a *triggering condition*. The triggering condition is typically separated from the code to execute, although it doesn't have to be. In all cases, the triggering condition is part of the specification of the behavior.

As said, it is important to note that the triggering condition is local to the behavior. This is an innovative aspect over existing systems. The code to be executed, and the conditions that cause it to execute, are brought together in the same place. This makes is significantly easier to reason about the behavior of the system, because one can reason over the functionality in isolation. In fact, this property is one of the key foundations on why it is so much easier to define complex systems using multi-agent systems than with traditional object-oriented technology. When describing the actions of the agent, one can describe its behavior based on its local perceptions of the world, without knowing what causes these perceptions and who is causing them. The agent just responds to the forces around him, which is a very natural model for many large scale problems in economy, computer science, finance, business, social sciences and many other fields.

The said triggering conditions use the events thrown by sensors and behaviors to drive its activation. A triggering condition conceptually includes two parts: a when-part that indicates upon to which events it responds, and an if-part that is based on values and other parameters to filter the events. This is another innovative aspect. Other languages, like Visual Basic, C# and Java, allow to throw and catch singular events. In RIDL, it is possible to use multiple events and complex conditions to select which events the behavior wishes to catch. In essence, the behavior applies some real-time form of *querying* on the events. This ability is a abstraction leap from the current languages.

It is possible to define triggering condition based on known agents. Since agents, like objects, are data types, it is possible to refer to them through the name of the variable that holds them (if held in a variable), of by reference (pointer). As a result, triggering conditions can be defined static.

Notice that we are waiting for a specific events. Firstly, that means that we need to specify which sensor of behavior we mean within the agent. Next, we need to define which type of event we are waiting for. A natural model to write this is to indicate the agent we are waiting for, followed by a dot, then name the sensor, followed by a dot, and then name the event. For instance "MyAgent.MySensor.updates" would be a natural model. However, the syntax could take any shape or form. The point is that we indicate the agent, then the sensor/behavior, and finally the event.

In the when-part of the triggering condition, events can be concatenated with "or" keywords, or by something with a similar meaning. This allows to create a list of events that we want the behavior to be sensitive to.

In the if-part of the triggering condition, all normal programming operators can be used, including boolean operators. In the if-part, all variables and operators that one could use in body of the behavior/method, are available. For instance, every variable within scope can be used, where the scope is defined as in traditional languages like Java and C#.

Any Event Operators

The explicit indication of the event we are waiting for is often not flexible enough. What if the developer wants to wait for multiple events? One solution is to exhaustively specify all possibilities, but in combination with other operators defined below, this can be very tedious (and sometimes even impossible due to lack of information). To create a more elegant solution, another innovative aspect is introduced: the generic event. The idea is that we will be waiting for any event coming from the named agent's sensor or behavior. One way of writing this is to omit the event name, another is to use the name "event" for the generic event, and many other notations can be conceived. For example, "MyAgent.MySensor" or "MyAgent.MySensor.event" would respond to both *changes* and *updates* events.

Any Sensor/behavior Operators

But what if we don't know which sensor or behavior we are waiting for? Maybe the agent's structure is totally unknown to us and we don't know which behaviors or sensors are inside the agent. It may still be that we wish to be informed about any activity within any behavior or sensor of the agent. To this purpose, we introduce a new innovative aspect: we introduce keywords such as "sensor" and "behavior", that replace the indication of a specific sensor or behavior. Again, other syntaxes can be used, but the point is that a stub is used instead of the explicit name of the sensor or behavior. A possible way of writing this is "MyAgent.sensor.changes". This will wait until any sensor in the agent MyAgent changes its values (it will ignore updates that do not change the value). Notice that again the developer has the choice of specifying the event, or use the

generic event indication (as defined in the previous paragraph). In the same vein, agents can replace the name of an explicit behavior with a stub that indicates we are waiting for any behavior. For instances, "MyAgent.behavior.activates" would mean that we wait for any behavior within agent MyAgent to activate, which would effectively allow us to monitor if the agent is active without having to know which behaviors are defined.

The ability to stub the names of sensors and behaviors will prove to be very important to write agent applications that can deal with other agents that are not yet defined at the time of writing of the first agent. However, it may present a new problem. The new construct works fine in the when-part of the triggering condition. However, how about the if-part of the condition. It may well be that we wish to respond to any sensor that has a value higher than 100. Because we stubbed the name of the sensor, the if-part is at a loss on which sensor it should test to see if its is over 100. What the if-part needs, is to know *which* sensor of the agent threw the event. To this purpose, yet another innovative aspect is provided. In the when-clause, we will annotate the sensor with the name of a variable. This variable will store a reference to the sensor that threw the event. Because the if-part is evaluated *after* the when-part, the if-part can use this variable to identify the sensor, and to investigate its properties (such as its value).

Within the body of the behavior that was activated by the triggering condition, the variable from the when-part can also be used. Therefore, the behavior can target its response to the event-throwing sensor, if needed. This is particularly powerful in combination with introspection attributes.

Any Agent Operators

A behavior may wish to respond to events coming from any agent, including agents that join the agent society at any point in time after the behavior started to wait for its triggering conditions. Indeed, even while the behavior is passive and idle, and without any code on the developers side, the behavior wishes to be aware of every agent in the system, including new joiners and agents that leave the system.

Notice that this problem is fundamentally different from the previous problems. An agent is defined at design time, and hence an agent class cannot change its definition at runtime. As a result, at compile time, an agent always knows the list of its own behaviors and sensors. Since we didn't use an *any* operator at the agent level, we always knew what agent we were talking about. Hence, the any operators at the level of events, sensors and behaviors can be resolved by the compiler if all agents are compiled together (and since we don't use the any operator on agent, and hence always refer to a variable or pointer with a *known* data type (agent), the compiler always knew).

The Any operator at the Agent level introduces a completely new level of complexity. If any operators on agent level are used, it is possible that agents join the society that were unknown at compile time. This implies that the agent must now resolve their any operators *at runtime*. This is no minor challenge, since RIDL is also is optimized for execution speed and need to minimize overhead. The innovative aspects that make it possible for RIDL agents to join societies coming from elsewhere are discussed in the section on "runtime level aspects" below.

So how will we allow agents to talk about agents they don't know? In the triggering condition of a behavior, instead of referring to a specific agent, one can also refer to an agent class. As was said before, an agent is a data type and as such has a name. The instances of the data type may or may not have names (variables versus dynamic memory allocation). So, by using the name of the agent data type (the name that is used in the agent definition), all agents in the class can be indicated. When a triggering condition uses the class name of the agent, it actually means that it waits for a specific event from any agent that is member of that agent class.

Again, while this works for the conceptual when-part of the triggering condition, it may present a problem for the if-part, and also for the execution of the code. When we pick up an event, then we may want to check if the agent that sent the event conforms to specific conditions (the if-part of the triggering condition). Therefore, when we receive an event, we also need to capture a reference to the agent that sent the event.

We'll take a similar approach as before, and we'll annotate the agent class with a variable name. This variable will store a reference to the agent that caused the event. Through this reference, we can access all public properties of the agent, including the values of public sensors. All actions normal on agent references can be done on these generated references.

There is another innovative aspect that is useful to complete the flexibility of the event mechanism. We want to be able to interact with agents that we know nothing about. To this purpose, we'll allow references (pointers) to the generic type *agent*. We'll also allow agents to be defined of this generic reference type.

When we write "Agent.sensor.event", then we'll respond to any event of any sensor from any agent. Notice that again different syntactical notations are possible.

Agent level events and operators

So far, we have assumed that only sensors and behaviors throw our type of events. However, agents themselves can also throw events. In particular, events that indicate that they are born, are dying, are joining the community (but were born elsewhere) or leaving the group (but not dying). This allows agents to respond even more targeted to each others actions. "Welcoming committees" can respond to agents joining, to inform them of the rules of the community, for instance.

Clearly, agent level events make most sense in combination with any agent operators. One possible notation is "<NewMember:>MyAgent.joins", which waits for any agent of the class MyAgent to join the group, and assigns a reference to that new agent to the variable NewMember. This is just a syntactical example, and the same effect can be obtained through completely different notations.

Another example is "<NewBorn:>Agent.born", which will respond to any agent of any type the is create new within the community. We'll talk more about communities when we talk about service level aspects.

No new agent level operators are introduced. Hence, the traditional terminate and new events correspond to dying and newborn agents. This fact is one of the reasons for the power of the previous innovations. Because the model expands traditional models so elegantly, all traditional object-oriented knowledge still holds. We'll talk about how agents can migrate when we discuss runtime aspects.

Subsume/resume

Killing agents is crude. Intelligent systems are often created in layers, where higher layers interact with lower layers and override lower layers. Nevertheless, the lower layers should remain active. Only limited functionality is overridden, while the bulk of the actions remain intact. Rather than removing the entire agent and replacing it by another, we also allow to subsume specific behaviors. Subsumed behaviors can later be resumed by other behaviors.

The subsumption/resumption of behaviors was first introduced by Rodney Brooks in 1986 in his paper "A robust layered control system for a mobile robot". It has been cited by

various authors writing on multi-agent systems. One innovative aspect here is that we fit it into the context of an agent description language, and in particular in the context where agents are "data types".

This is directly related to the innovation of introducing behavior-based multi-agent systems outside the domain of robotics. It ties in perfectly with the elegance of the presented approach, that brings simplicity to the most complex systems.

Subsuming and resuming are runtime features. They can be used on any behavior we can name. For instance, if an agent is stored in a variable "MyAgent", we can directly specify the behavior with the name "MyBehavior" through "MyAgent.MyBehavior.subsume". Other syntaxes can achieve the same effect, as we are not describing syntax here but capabilities.

A behavior can be subsumed by any behavior, including itself. A subsumed behavior can be resumed from any other behavior (hence not by itself, since it is inactive). The behavior that does the subsuming need not be the same as the behavior that does the resuming.

Every behavior has a predefined property called "subsumed" (or some equivalent name). Although the property is part of the behavior, it is a sensor. The type of the sensor is scalar (for instance "int" or "integer"). The property counts the number of times a sensor is subsumed. If a behavior is not subsumed, its subsumed property will be zero. Every time the behavior receives a subsume request, the counter will be increased by one. Every time the behavior receives a resume request, it will decrease its behavior. The behavior will work if the subsumed property is zero. A subsumed behavior completes upon subsumption! This means that upon resumption, it will re-evaluate its triggering condition. This implies that the behavior first looks at its environment when it is reactivated. This may prevent it from doing the wrong actions.

Notice that the subsumed property is a true sensor. Hence, when it changes, it throws an event. The subsumption status can be used in triggering conditions. In combination with a "completeWhen" statement (described later in this text), this also allows a behavior to monitor its own subsumption status. Using this construct, a behavior can execute code just before it is subsumed, ensuring that no damage because the behavior is interrupted in the middle of its body.

Because the features are runtime, they can also be done for agents of which we only have reference. Hence, these features can be invoked in the body of behaviors, who select other behavior with *any* operators. This advanced capabilities has never before been described, and is certainly one of the innovative aspects of this patent.

The features on subsumption and resumption have been described as syntactic keywords. However, these features can also be offered as methods that are by default available on the agent. For agent "shopAgent", if we want to subsume its "buyBehavior", we could write "shopAgent.subsume("buyBehavior")" if the functionality was implemented as a method on the agent. If subsume were a keyword, the same would be written along the lines of "subsume shopAgent.buyBehavior". Whatever the way of writing, the concept remains the same.

Inheritance

One of the key features of object-orientation is the capability to create derived types. An object's functionality can be refined by *inheriting* all of the functionality, and *overriding* functionality as needed.

We will do the same with agents. This capability is a direct and very elegant consequence of the approach to agents as data types.

When an agent is refined, one can do exactly the same as with objects. Methods can be overridden to be refined.

Behaviors cannot be invoked by other code, because they decide themselves when they are activated. As such, behaviors have no parameters. When behaviors are overridden, they are immediately replaced by new behaviors.

Sensors are a type of variables. Hence, normal scope rules apply. This implies that sensors replace sensors with the same name.

Most importantly, the event mechanism remains intact upon inheritance. When an agent is inheriting a behavior, the triggering condition of this behavior will take into account that it needs to look at the behaviors in the child agent. If no behaviors are present in the child, it will look for these behaviors in the parent agent. In combination with the any sensor/behavior operator, this allows agent to make very complex logic, where the parent can provide functionality such as persistence without explicitly knowing the structure of the child agent, amongst other capabilities.

Agent any operator revisited

In the definition of the agent any operator, we said we could specify the name of the class. If the any class specifies the name of a class that has children, all of its offspring will also be considered by the any operator. This makes sense, because the child of an agent is an agent of the same class, with refinements.
If the name of a child agent is used in the any operator, then the parent agent will not be part of the any operator. Again this makes sense, since the parent is not part of the class of the Child Agent.
As one example if a *car* is a child agent of a *vehicle*,

- Then when we speak about any vehicle, we also mean cars.
- But when we speak about cars, we do not mean any vehicle.

The fact that any operators are inheritance-aware makes them very powerful for complex decision making. Planning examples:

- An agent could activate a behavior on the following conditions: if any ship is approaching, and there no military ships in the neighborhood, then activate me.
- If the teacher doesn't have a classroom, and no science classrooms are available, then ...

Advanced capabilities

Notice that sensors can themselves be agents, since sensors are variables, and agents are data types. Also, agents can be passed as parameters to methods.

Splitting events from behaviors (event handling constructs)

We have talked about behaviors having triggering conditions. Clearly, it would also be possible to create a language construct that responds to triggering conditions, and that either throws a specific event, or that directly invokes a method.

Such a construct in one aspect essentially splits triggering conditions from the method/behavior, but achieves the same as we intended and is equally covered by our patent request.

Changing triggering c nditions

Similarly, triggering conditions could mingle the when and if part. It may be more elegant and purist to split the two conditions, but mingling the two in a single construct is also contemplated as part of the present invention.

SERVICE LEVEL ASPECTS

As integration of software gets ever harder, and software gets ever more complex, the way software is being designed is changing. Recently, there has been a trend toward service-oriented software engineering. The essence of this approach is that a software application has an interface based on standards, such as XML Web Services. The software offers its functionality as a service through this interface. Integrating different software packages becomes a mere matter of gluing the services together.

In software engineering, namespaces are used to group together objects into logical assemblies. A "disk" namespace could contain all routines that interface with the disk, for instance.

According to one aspect of the present invention, namespaces are used to bring together agents in a similar way as objects are brought together. In particular, agents can be part of the namespace, and can share a namespace with objects. In other words, in this aspect the language doesn't distinguish between agents and objects, they both follow the same rules.

At the level of namespaces, we introduce a new concept called a service. Services at the language level are similar to namespaces, in that they bring together objects and agents with similar functionality. In particular, it brings together agents and objects that jointly achieve a single service.

A namespace can be indicated to be a service by using a new keyword, by using an attribute that annotates the namespace, or by assuming that every namespace that contains an agent is a service.

If a namespace is a service, it offers functionality. The functionality can be accessed through a defined interface. The present invention provides two new ways of specifying this interface.

The first method is to explicitly define an object or agent to be the interface to the service. This can be done by providing an attribute to the object or agent.

The second method is to name the object or agent identical to the name of the service.

In this case, the public variables and methods are the actual interface of the service. The agent or object of this class will be instantiated automatically when the service is started, and only one instance of this agent or object can be created per service.

AGENT ORIENTED DATABASE LEVEL ASPECTS

The latest databases, such as Microsoft SQL Server "Yukon", have advanced programming languages at the record level (in Yukon's case, e.g. C#).

ASPECT 1

One approach to agent-oriented databases (AODB) is to consider every record (object in an OODB) as a special kind of agent. The agent contains only sensors (no behaviors), and inheritance on these agents is forbidden. Every externally field, and every calculated or otherwise obtained field, is considered to be a sensor. Hence, a database is a set of agents that have only sensors.

This approach monitors what happens to the fields of every record. Every time a field is updated, an updated *event (of the type that is used by a behavior's triggering condition)* is thrown. If a field changes value, a changes event is thrown.

The result is that behaviors can be defined that monitor and respond to changes in the database. From a conceptual point of view, the database is filled with agents, to which other agents can respond.

ASPECT 2

Alternatively, a database agent could be just like any ordinary agent, with sensors and behaviors and other attributes. The difference between a database agent and an ordinary agent is that its sensors are stored in the database. In this case, the software designer conceptually works with full blown agents.

In this case, the structure of the database and the structure of the agent system must sufficiently match.

The advantage of this approach is that the designer has the complete freedom of the agent model. The compiler will create the tables needed to support the model.

One drawback of this approach is that legacy databases with the wrong structure might not be able to be used. This approach doesn't match well with pre-defined databases.

When using this approach, it is desirable to annotate an agent with a keyword to indicate that this agent is persistent. This allows the compiler to distinguish between persistent and non-persistent agents.

RUNTIME LEVEL ASPECTS – MOMENT OF EXECUTION

Hierarchy graph shows which behaviors to execute

The power of the event mechanism described in the compiler aspects is shown by the type of optimizations the compiler can do. These optimizations have a profound impact on the runtime performance. They are here classified under runtime, but they require the compiler to take action to generate the necessary lists and other materials for the runtime engine.

Real-time optimized and speed-optimized execution is one of the key features of RIDL.

As said, a triggering condition is split into a when and an if part. The when part specifies the events that trigger the evaluation of the if part of the triggering condition.

Since every event is linked to a previously defined behavior or sensor (and a fortiori previously defined agent), and since every behavior relies on these events in its triggering condition, a dependency graph can be drawn between behaviors, with sensors as leaves in the graph.

Every time a sensor is updated or changed, or a behavior is activated or completed, the event propagates through the graph and sets flags of triggering conditions that need to be re-evaluated.

The triggering conditions are evaluated and if they are met, then a flag is set to indicate that the behavior needs to be executed.

There is a pool of execution threads that selects a flagged behavior, and executes it. The pool of execution threads could be a single thread on the one extreme, or as many threads as there are behaviors at the other extreme. The number of execution threads is dependent on the compiler execution, but could be decoupled from the number of agents and behaviors in the system.

Notice that "flagging" a behavior could take many forms. It could be a variable set as a flag, or the behavior (id) could be added to a list, and in the extreme case a execution thread dedicated to the behavior could start (which is equivalent to flagging and starting the execution at the same time).

<TODO: DRAWING
List of all behaviors
List of triggering conditions that need to be re-evaluated sorted by priority
List of behaviors that need to be executed sorted by priority
List of sensors
A pool of one or more threads that continuously evaluate the triggering conditions
A pool of one or more threads that continuously that the behavior with the highest priority that needs to be executed.

For every sensor
List of behaviors to re-evaluate triggering conditions on value change
List of behaviors to re-evaluate triggering conditions on value update

For every behavior
List of behaviors to re-evaluate triggering conditions on activation
List of behaviors to re-evaluate triggering conditions on completion
/>

Automatic priority detection for behaviors

According to one aspect, behaviors that are lower in the hierarchy graph are given higher priority. Indeed, being lower in the hierarchy means that the behaviors are closer to the hardware or software interface. That means that they are closer to the events, and that they require to be more responsive.

An example in robots and machine control makes this clear. If a behavior is directly coupled to a hardware sensor, it is likely that very rapid response is needed. However, at the highest level, behaviors respond to sensors that were built by behaviors already responding to other RIDL sensors. In other words, the behaviors that are higher in the graph work with more abstracted data. Processing this information is usually less time sensitive than the low level behaviors ("reflexes" versus "thoughts").

As before, an event of a sensor or behavior will propagate through the dependency graph, and will put the triggering conditions that need to be re-evaluated in a list. A triggering condition processor will process this list. The list is priority-based, which means that every time a new behavior's triggering condition is added to the list, it will be sorted into the execution queue such that it is executed as soon as it has highest priority of all waiting behaviors. The priority reflects the distance to the leaves, where a leave has highest priority, and every additional dependency reduces the priority.

In smaller systems, it is possible to evaluate the triggering condition immediately, which is the extreme case. In this case, every behavior automatically has highest priority, since no other behaviors are waiting since evaluation is immediate.

If the triggering condition is met, the behavior is stored in a new list which contains the behaviors to execute. Again, we can keep this list sorted by the priority of the behavior (with the same priority definition as above). The pool of threads will always execute the waiting behavior with the highest priority.

The net result is that lower level behaviors may execute multiple times before a higher level behavior that is dependent on them gets the chance to execute. This means that higher level behaviors can miss "frames", where a frame is defined as an triggering condition that would have been true, had it been evaluated. Although this may seem strange, this is actually beneficiary to the software. It assures prompt responses at the lowest level, where prompt responses are needed. At the same time, it matches with the uncertainty principle that governs the actions of agents. An agent is never sure that what he believes to be true, is actually (still) true. As such, an agent is required to keep checking if it assumptions still hold. The net result of this uncertainty principle is that multi-agent systems are much more robust, because they are built from the ground up to handle anomalies.

A behavior is often waiting for multiple events. In this case, the behavior will always have a priority that is one lower than the event that just caused it to execute. Hence, the priority of a behavior changes dynamically at runtime.

An event of a sensor has a priority this is one lower than the behavior that updated the sensor and caused the event. In case the sensor is updated from outside a behavior, it will be considered as a leave event, with the highest priority.

A behavior that is triggered on system-defined events, such as timers, will be considered as leave behaviors and will have the highest priority.

There are two more rules that need to hold on all behaviors, and that impact the priority assigned:

- A behavior that is activated based on the completion of another behavior, is always of lower priority than that other behavior;
- A behavior that is activated based on the activation of another behavior, has the same priority as that other behavior.

Exhaustive behaviors

Sometimes the above scheme may result in wrong priorities. In particular, there can be two graphs of dependencies inside the software. One of the graphs is responsible for executing some work. The other graph is merely monitoring the actions of the other graph. Because they start from the same sensors, and do not interact at the higher level, the compiler cannot assign a higher priority to one graph or another. It assigns similar priorities to both, making them compete for resources at runtime.

A natural approach is to allow the user to explicitly define the priority of the behaviors. However, this is very error prone. Therefore, another method is provided that is safer and more intelligent.

When a behavior is marked as "exhaustive", this means that the behavior is not allowed to miss any frame. That means that the behavior will respond with sufficient priority to ensure that it is executed before the next time its triggering condition becomes true. Exhaustiveness does not provide guarantees that the behavior is executed within a

certain timeframe. It merely guarantees that every time the triggering condition becomes true, it will be executed, and that the next execution will occur after the previous one has completed.

An exhaustive behavior is exhaustive relative to its own events. Its exhaustiveness has no impact on other behaviors in the dependency list. In other words, it is not because a behavior is exhaustive, that another behavior that it is waiting for, also becomes exhaustive or is in other ways changed in priority. Only behaviors explicitly marked as exhaustive are sure never to miss any frame.

Of course, if another behavior relies solely on an exhaustive behavior, then it may be triggered more frequently than the normal behaviors, because a lot of events it is waiting for are generated.

The agent designer must be very careful when using exhaustive behaviors. Because they have a higher priority, they may cause starvation amongst ordinary behaviors. In particular, if an exhaustive behavior directly or indirectly triggers its own triggering condition, this will result in a continuous activation of the exhaustive behavior. Since it has high priority, it will immediately execute and trigger itself again. The net result is something similar to the system "hanging".

One way an exhaustive behavior may accidentally trigger itself, is when it uses an *any* operator in its triggering condition that also includes sensors/behaviors that are updated/triggered through its own actions (directly or indirectly). This may potentially be hard to catch.

Redundant behaviors

Redundant behaviors are the opposite of exhaustive behaviors. When a behavior is marked as redundant, this means it has a lower priority than all normal behaviors.

Like exhaustive behaviors, redundant behaviors do not change the priorities of other behaviors in any way. Only behaviors explicitly marked as redundant behaviors have this lower priority.

Of course, if another behavior relies solely on a redundant behavior, then it will never be triggered more frequently than the redundant behavior, because no events it is waiting for are generated.

Real-time mapped hierarchy graphs

Finally, in support of real-time systems, behaviors are allowed to be explicitly mapped to specific numeric priority levels. The developer can fix these behaviors. For every behavior where no numeric priority level is defined, the formerly defined rules apply.

It is preferred that developers avoid numeric priority levels, as they potentially have a very disturbing impact on the execution of the agents. Only in hard real time applications, where the designer is fully aware of the implications of his actions, should this feature be used.

continueWhen statement

The continueWhen statement is followed by a triggering condition. It is a normal statement that can be used at any point in the *body* of a behavior. It will instruct the behavior to wait until the specified triggering condition becomes true. This statement is particularly useful if a behavior needs to execute a sequential series of actions. It also provides a basic construct to ensure synchronization between behaviors.

An example is when a robot needs to raise it arm (action) until it reaches a certain height (sensor), with some additional complexities. After is has done so successfully, it needs to push a button.

The continueWhen statement is a shorthand notation for functionality that could also be implemented using a state machine. The behavior that contains the continueWhen statement can be split into several behaviors which use a state machine in combination with the specified triggering condition to have the same effect. So, initially the state machine is in its state 0, and waits for a state 0 together with the triggering condition of the behavior. Upon completion of the first part of the behavior, it puts the state machine in state 1. A second behavior, which models the part after the continueWhen, waits for the state 1 together with the triggering condition specified in the continueWhen. When the last behavior in the state machine has been activated, the state is put back to 0.

In one aspect, the compiler will apply just this transformation to the software. The result is that the very behavior includes a triggering condition and a method that is executed to its endpoint, which makes scheduling easier. Here's an example:

```
void MyBehavior() : behavior
when TrigCondWhen
if TrigCondIf
{
        Statement1;
        Statement2;
        continueWhen TrigCondContinueWhen1
                if TrigCondContinueIf1;
        Statement3;
        continueWhen TrigCondContinueWhen2
                if TrigCondContinueIf2;
        Statement4;
}
```

This can be transformed into:

```
int MyBehaviorState = 0;

void MyBehaviorPart0 () : behavior
when MyBehaviorState.changes or TrigCondWhen
if (MyBehaviorState == 0) and TrigCondIf
{
        Statement1;
        Statement2;
        MyBehaviorState = 1;
}

void MyBehaviorPart1 () : behavior
when MyBehaviorState.changes or TrigCondContinueWhen1
if (MyBehaviorState == 1) and TrigCondContinueIf1
{
        Statement3;
        MyBehaviorState = 2;
}

void MyBehaviorPart2 () : behavior
when MyBehaviorState.changes or TrigCondContinueWhen2
if (MyBehaviorState == 2) and TrigCondContinueIf2
{
        Statement4;
        MyBehaviorState = 0;
}
```

The continueWhen statement allows the developer to describe the problem in a more natural sequential way instead of with state machines. However, one notable difference between a state machine and a behavior using continueWhen statements, is that variables local to the behavior can be used before and after the continueWhen statement without losing their value. If the agent designer were to use two distinct behaviors, he would have to use a variable or sensor global to the agent to achieve the same goal. Using the continueWhen statement allows the designer to use a local variable to achieve the same result.

The compiler will transform the continueWhen behavior into multiple behaviors, and will ensure that variables are accessible from both behaviors and not from anywhere else (since they are conceptually local). It will achieve this by creating variables global within the agent, with unique names that are not referenced anywhere else in the agent.

In RIDL2004, the continueWhen statement is called the *resumeWhen* statement. As before, these little syntactical variations are not relevant, this text highlights the principles behind the statements.

completeWhen statement

The continueWhen statement is used within the body of a behavior. However, both the when and the completeWhen statements are used outside the body of the behavior. As said, the when condition indicates when the body is executed.

The completeWhen statement is the inverse of the when statement. It specifies a triggering condition upon which the behavior should *stop* executing. The completeWhen may again be followed by a body. When the completeWhen is triggered, the body of the behavior is stopped, and the body after the completeWhen is executed. Within the body of the completeWhen, all local variables defined in the body of the behavior can be accessed. Conceptually, the code is inside the body and replaces all that remains to be executed.

The completeWhen statement can be achieved by transforming the body of the behavior. Assume we have following behavior:

```
void MyBehavior() : behavior
when TrigCondWhen
if TrigCondIf
{
        Statement1;
        Statement2;
        Statement3;
}
completeWhen TrigCondCompleteWhen
if TrigCondCompleteIf
{
        CStatement1;
        CStatement2;
}
```

Then this can be converted into following code with the same effect:

```
bool MyBehaviorCompleteNow = false;

void MyBehaviorMustComplete() : exhaustive behavior //& highest priority
when TrigCondCompleteWhen
if TrigCondCompleteIf
{
        MyBehaviorCompleteNow = true;
}
```

```
void MyBehavior() : behavior
when TrigCondWhen
if TrigCondIf
{
        if not MyBehaviorMustComplete
        {
                Statement1;
                if not MyBehaviorMustComplete
                {
                        Statement2;
                        if not MyBehaviorMustComplete
                        {
                                Statement3;
                        }
                }
        }
        if MyBehaviorMustComplete
        {
                CStatement1;
                CStatement2;
        }
}
```

The check on the completion condition must have the highest priority, because no matter how high the priority of the behavior, the fact that it must complete has even higher priority. The use of the exhaustive keyword is safe here, because it does not update anything that triggers an event. It updates a variable, not a sensor. The compiler can safely convert the first code into the latter, and thereby implement the functionality.

This is not the only way of implementing the required functionality. Instead of continuously nesting if statements, one can also throw an event in language is C# and Java, which can be caught at the end of the method. This may be more efficient in execution, especially with the way events are defined in C#, and provides an alternate way of implementing the keyword.

Whatever method is chosen, the compiler can transform a behavior with a completeWhen statement into a behavior without such a statement. As such, it is merely a very useful and powerful shorthand notation. The use of the statement is extremely frequent, especially in combination with the system-defined subsumed sensor of every behavior, as shown in the next example:

```
void MyBehavior() : behavior
when OtherSensor1.changes
{
        Statement1;
        Statement2;
}
completeWhen MyBehavior.subsumed.changes
if MyBehavior.subsumed
{
        Clean_up_behavior;
}
```

In fact, the use of this statement is so frequent that it is allowed to use multiple completeWhen statements at the end of a behavior, to catch different events and take different actions. Should you wish to take the same action on multiple events, then you simply make the triggering condition more elaborate.

After transformation, the compiler will generate code that uses if statements to execute the multiple completeWhen statements. In the example with the nested if statements:

```
void MyBehavior() : behavior
when TrigCondWhen
```

```
if TrigCondIf
{
        Statement1;
        Statement2;
        Statement3;
}
completeWhen TrigCondCompleteWhen1
if TrigCondCompleteIf1
{
        C1Statement1;
        C1Statement2;
}
completeWhen TrigCondCompleteWhen2
if TrigCondCompleteIf2
{
        C2Statement1;
        C2Statement2;
}
```

This can be converted into following code with the same effect:

```
int MyBehaviorCompleteNow = 0;

void MyBehaviorMustComplete1() : exhaustive behavior (& highest priority)
when TrigCondCompleteWhen1
if TrigCondCompleteIf1
{
        MyBehaviorCompleteNow = 1;
}

void MyBehaviorMustComplete1() : exhaustive behavior (& highest priority)
when TrigCondCompleteWhen2
if TrigCondCompleteIf2
{
        MyBehaviorCompleteNow = 2;
}

void MyBehavior() : behavior
when TrigCondWhen
if TrigCondIf
{
        if not MyBehaviorMustComplete
        {
                Statement1;
                if not MyBehaviorMustComplete
                {
                        Statement2;
                        if not MyBehaviorMustComplete
                        {
                                Statement3;
                        }
                }
        }
        if MyBehaviorMustComplete == 1
        {
                C1Statement1;
                C1Statement2;
        }
        if MyBehaviorMustComplete == 2
        {
                C2Statement1;
                C2Statement2;
        }
}
```

Again, a similar effect can be obtained by throwing events instead of using nested if statements. The compiler could also include check more intelligently, to reduce the number of tests that need to be executed. The above only gives the basic principles.

Notice that subsumption also works the same way. Through the use of if-statements, or by throwing events, the method that is subsumed is immediately terminated without actually killing the thread (the latter cause more overhead processing and more complexity).

The above transformations show that the runtime level execution, discussed above for behaviors without continueWhen and completeWhen statements, can also be applied to behaviors with these statements.

Deadlock Detection

Different types of deadlocks can occur. The detection described here is only valid for one type of deadlock: the case where behavior 1 waits only for behavior 2, and behavior 2 waits only for behavior 1. This type of dependency generates a closed loop in the dependency graph, and will result in an error at compile time. In multi-agent systems, behaviors usually wait for sensors, and sensors can be updated by many sources, which makes it impossible for the system to understand when it is truly deadlocked. The prudent developer is advised to create time-outs, through watchdog behaviors or through the use of abort-if statements.

RUNTIME LEVEL ASPECTS – MOBILITY OF AGENTS

Communities

We have talked about agents as if they are all aware of each other. However, according to one aspect, an agent is only aware of the agents which are present in its *community*.

In previous parts of the text, a community typically is the same as an application. All agents within an application are usually aware of each other.

Notice that an application that contains multiple communities is generally considered to be multiple applications. Indeed, this is what service oriented software engineering is all about. A community is implemented as a namespace or as a service.

Communities themselves can migrate. That is, a service can copy itself to another computer, and can start itself remotely. As a result, the community is now also active on the other computer.

An agent can migrate between replicated communities. In particular, an agent can generate a message that is sent to the other community, that contains its status. This includes its type and the value of all of its sensors. A proposed way of implementing this is to "dehydrate" the sensors of an agent into XML, and send this XML definition to the replicated community. There, a new agent is created and all sensors of the agent are set to the values received in the XML message ("hydrate"). The new agent sends an acknowledgement back a message to the first agent, who can then choose to destroy himself. If the agent doesn't destroy himself, he has simply replicated himself.

When an agent is newly created at any time and for any reason within a community (e.g. also through the *new* operator active on the data type), then the triggering conditions of all behaviors will be evaluated. Depending on the result, the behavior will activated or not.

It is therefore quite possible that a behavior that was activated in a community, and that spawned the migration of its agent to a new community, is not immediately activated within the new community because its triggering condition isn't met in the new community. Hence, it is unsafe to rely on any actions taken by the agent of the new community immediately after its creation, short of actions taken through the constructor of the agent. Like objects, agents have constructor and destructor methods.

Grid computing

Communities are very powerful, and their ability to spawn copies of themselves (on other processors and machines) allows them to take advantage of all authorized processing power that is available in a network. This is our multi-agent interpretation of grid computing. All computers nearby that are authorized for use, will automatically and dynamically create a computing grid that executes the multi-agent system. This allows the multi-agent system to grow far beyond the capabilities of a single computer. The advantages for distributed computer are obvious and large.

As an example, consider a video game. There are always a lot of sidekicks you can use in a game. Assume you have an army at your disposal, where every soldier is an agent. The more powerful your machine is, the more agents (soldiers) you can command. However, if you have another PC nearby, make sure you put him in your LAN, and you will automatically be able to use that additional PC to control many more agents on your computer. For the gaming industry, such innovation would be revolutionary with no effort on the programming side. Indeed, time-to-market is one of the key drivers behind games, that always aim to improve on the last most popular game in their genre.

The same is true when it comes to physics calculations, large banking calculations, and other scaled-up applications.

All these abilities come to multi-agent systems as part of our implementation of communities.

Migration between similar communities

Definition: a community is structurally identical if the definitions of all agents are identical, and no community has additional agents relative to the other. In previous sections, we have been talking about structurally identical communities.

Definition: the signature of a sensor is the name of the sensor, along with its type.

Definition: an agent A is considered similar to agent B, if and only if it has the same name, and if all of the sensors of agent A have a sensor with the same signature in agent B.

Definition: a community C1 is a similar to community C2, if and only if some of the agents of C1 are similar to agents in C2. Not all agents of C1 need to be similar to agents in C2.

An agent can migrate between similar communities. In particular, an agent can generate a message that is sent to the other community, that contains its status. Its status is composed of the name of the agent, and for every sensor of the agent, the signature of the sensor and its value. This information is assembled in a data package (e.g. in XML definition), and sent to the other community with the request for replication.

In the receiving community, the specified name of the agent is looked up. A check is performed if this agent is similar. If it is not, then a message in that sense is sent back and no further actions are taken. If the agent is similar, a positive acknowledgement will

be sent back to the sending community and a new agent instance is create in the receiving community. All the received values for the sensors are assigned to the agent. The creation of the agent, and the updates of the sensors, will send a number of events through the receiving community.

Clearly, the migration between structurally identical communities is a subset of migration between similar communities.

How an agent initiates migration

Every community is a service or has an interface. This interface accepts messages, such as ACL (Agent Communication Language) messages or XML messages. RIDL usually works with XML Web Services to establish communication between communities. Every community has a unique ID, usually represented by a URL.

An agent that wishes to migrate, needs to know the ID of the community to which it migrates. Two special functions are pre-defined methods of every agent:

- int copyToCommunity(ID)
  The community with the specified ID is sent the data package described above, and asked for replication. The return code of the function contains an indication of success (0 = copy successful).
- int migrateToCommunity(ID)
  Here, first a copyToCommunity will be executed. On success, the agent that was copied is killed. If an agent migrates itself, then it may be that it doesn't execute the line after the migrate instruction. If the migration is not successful, the next line will always be executed, allowing diagnosis based on the return value by the agent.

Again, the syntax of the implantation may vary. Important in this aspect is that the developer need to know only the ID of the similar community. He needs no knowledge of communication protocols, Web Services, ACL or any other technology. Whenever an agent is created, it has the ability to copy and migrate across similar communities, without any work by the developer. The functionality is here specified as a method that is always available on every agent. An alternative approach is to provide a library that contains following functions:

- int copyToCommunity(AgentType, ID);
- int migrateToCommunity(AgentType, ID);

A key point of this and previous sections is that migration is native to the language, either in a pre-defined method per agent, or in a library that is delivered together with the language. By changing the way a community starts, the developer can quite easily use the same source code to create primary and secondary communities. By primary community, we mean a community that creates its own agents in some sort of bootstrap. By secondary communities, we mean communities that are similar to the primary one, but contain no agents and are installed on other PC's on the network, waiting to receive agents that copy or migrate to them.

Any primary community can be turned into a secondary community automatically by analyzing the constructors, and removing any statements that create the initial agents.

Quite clearly, this is a very powerful concept. Game developers can provide a runtime engine that the gamer can install on PC's in the LAN, and the game can immediately exploit the power of those PC's, when written in RIDL. The game writing company doesn't need to do anything to make this happen.

Looking for communities with similar agents

How does the agent obtain the ID of the community to which it wishes to migrate? Here, we'll use the infrastructure already present within the definitions of ACL and within the definitions of XML Web Services.

However, what we teach here builds on those systems to go further. Sometimes an agent is looking for all communities that are available on the computer or on the network. But usually, it is specifically looking for communities that contain a similar agent. In general, we assume that when an agent is interested if there are communities out there, it knows with which type of agent it wishes to speak. Therefore, an advanced query is provided, which will look for all communities in the neighborhood that contain a specified agent. The agent specified may be the agent itself, but it may also be another agent. The lookup could be for similar agents as well as for structurally identical agents. Finally, it would also be nice to have an idea of how many instances of the agent of the specified agent signature are already present in the community.

It is desirable to allow a designer to use all this functionality, without knowing anything about the protocols behind it. It should be completely transparent, and fit in with the rest of the model in a very natural way.

Every agent will contain a pre-defined method, of the following format:

- communityCollection findCommunities(StructuralIdentical : bool = false)

The communityCollection type is a collection of communityCollectionItem.
The communityCollectionItem is a structure with two parts: the ID of a collection, and the number of instances of the agent that is already present.

The parameter "StructuralIndentical" indicates whether we want to look for structural identical agents (true) or similar agents (false). The default value is to look for similar agents.

Again, this functionality could also be part of a library delivered with the language, where a function such as the following is available:

- communityCollection findCommunities(AgentType : agent; StructuralIdentical : bool = false)

Of course, the identical function can be obtained by changing the syntax in a number of ways. What we intend here is the concept of how a generic function allows an agent to understand which communities are within reach, and how many of "its kind" are already in these communities. Based on that information, the agent can decide if he wishes to copy himself or migrate to another community. To do all of this, the developer needs to know nothing about advanced protocols.

Agents that work across communities

In the previous sections, we have taught a way an agent can migrate from one community to another. We have also taught a way to identify which communities have an structurally identical agent, and how many instances of identical agents they have. We have assumed that the developer designs agents to migrate in pre-defined ways.

In this section, we introduce agents that automatically migrate to communities either less agents of a particular type, or with more remaining CPU power. There is one significant problem with agent migrating from one community to another. The any operators are community dependent, and will not pick up agents in other communities. Let's illustrate this with an example.

Suppose we are in a multi-user video game, and you have an army of 1,000 soldiers fighting for you. Each of these soldiers is represented by an agent with complex fighting and psychological behaviors. Suppose you have another powerful computer in your local network. If you could use that machine in your game, maybe you could have an army of 10,000 or more soldiers at your disposal. You would have more power in the game. Fortunately for you, the game designer made this possible because he also provides a separate application that can be installed on the other PC, and that contains a secondary community, ready to receive your soldiers.

However, soldiers who are in the secondary community, and who respond to their environment through any operators, can no longer see the other agents that are in the primary community.

The model which we have defined in the previous pages holds a solution to this problem. The entire model is driven by the events that are created by sensors and behaviors. The state of an agent is largely stored in the values of the sensors. To have agents respond to each other across communities, we need stubs that hold the sensors. It is not needed to exchange the events of behaviors, because behaviors should always be private to an agent. An agent should never rely on behaviors of other agents. An agent should always express its entire public status through public sensors and public methods (in to a lesser degree with public variables).

We would like these stubs to be created automatically by the system without much intervention from the designer. However, we do wish to give the designer the freedom to decide which agents can move out to other machines, and which cannot.

To accomplish this goal, an agent can be annotated as "autoMigrate" with a keyword or attribute, for instance:

```
void MyBehavior() : autoMigrate behavior
```

For all autoMigrate behaviors, the compile will also create a second "stub" agent in the primary community that contains all of the sensors, but none of the behaviors. This stub agent will later be responsible to transfer sensors, as described below. In the secondary community, the full definition of the autoMigrate agent will be available, as well as stub agents for every agent that the autoMigrate agent relies upon.

When an autoMigrate agent is instantiated, it works like an ordinary agent. The system will monitor the execution list. The execution list is the list of behaviors that are need to be executed, sorted on priority, as explained in the section on automatic priority detection.

If the list of behaviors gets longer than X, then an autoMigrate agent will be selected for migration. X is a parameter that can be configured by the designer. So, we use the event model and the resulting priority detection as a measure for the busyness of the computer's processors. When parameterized right, this allows the autoMigrate agents to stay on the computer as long as there is processing power.

To select an autoMigrate agent for migration, several criteria can be used. For instance, the autoMigrate agent that relies on the smallest number of external sensors can be chosen. In this case, the sequence in which autoMigrate are selected for migration can be determined at compile time.

Another selection method is to use the autoMigrate agent that is most frequently activated, since this will reduce the workload of the current computer most. This is best determined at runtime.

Or an autoMigrate agent may be chosen that has the largest number of behaviors on the execution list.

Finally, a random autoMigrate agent may be chosen.

More methods are easy to come up with. Whatever method is used, one of the autoMigrate agents is selected. The agent will migrate to the secondary community in the same way as described in earlier sections.

In the primary community, every time an sensor is updated in an agent that the autoMigrate agent relies on, the value update for the sensor is sent to the secondary community where it updates the stub for this agent.

In the secondary community, every time a sensor of a migrated agent is updates, the related value of the sensor is sent back to the primary agent, where it updates the stub for the migrated agents. Because we are updating a sensor, this will recreate the events that existed in the secondary community. The compiler will ensure that the events created by the stub are to the using agent identical to the events created by the original agent.

The sending back and forth of information can for instance be done through XML packages. System defined behaviors could be added to the model, that respond to the events of the agents, and send the information to the stub agents in the other community. Agents either send the name of the stub to the other community (with known ID), or to a unique ID (pointer) directly to the stub's sensor.

At design time, we specify a class to be autoMigrating. However, it is the individual instances that migrate. Every agent will decide for itself when it migrates. Hence, some of the instances of an autoMigrate agent may have migrated, while other instances of the same agent class have not migrated.

If more computers are in the network, there may be multiple secondary communities for a single primary community. This was also pointed out in the section on grid computing. On each migration, the system can choose a community in the same way as said before in the section on looking for communities with similar agents. Here, we will be looking for communities with agents that are structural identical.

Although only public sensors of agents are send back and forth, autoMigrate agents can generate a lot of network if used unwisely. This is why we allow the designer to specify which agent is autoMigrating, and which are not.

Notice that the outlined principles are especially useful in grid computing and video games.

Security issues for migrating agents

Security issues are resolved at the level of security for XML Web Services (or ACL level). The security measures on the use of services should prevent agents from migrating to communities for which they do not have authorization.

DEBUG LEVEL ASPECTS

Debug behaviors

Multi-agent systems are very hard to debug, because they have so many parallel behaviors. Runtime errors can be generated by racing conditions, and co-occurrences of

events that are extremely hard to recreate. "Stepping" through code rarely makes sense, because the side effects make the recreation of the concurrent nature of the system incomplete. Traditional debugging methods break down when debugging across agents. Therefore, an agent designer should ensure that his agents are as robust as possible to external errors.

To help the designer, we can *exploit the features* of the concepts outline above to the benefit of the designer. We effectively take debugging to the next level, by reasoning in terms of agents; behaviors and triggering conditions rather than in terms of methods and sequential code.

We introduce a new keyword or attribute that indicates that a behavior is a "debug" behavior:

```
void MyBehavior() : debug behavior
when TrigCondWhen
if TrigCondIf
{
        // statements
}
```

Debug behaviors are treated different from ordinary behaviors.

First, debug behaviors are only compiled when the system is in debug mode. In release mode, these behaviors are automatically removed.

Second, debug behaviors should always have the highest priority. The designer should minimize the statements in the body of debug behaviors. Most often, the behavior has an empty body, and the mere triggering of the behavior is what we're looking for. We can put a breakpoint on this debug behavior.

Third, a debug behavior cannot have continueWhen or completeWhen statements. It cannot be subsumed or resumed. In general, a debug behavior does not participate in activities of the agent system. It merely observes (and sometimes logs).

Fourth, in debug mode, every behavior will continue logic to enable a *freeze* of the system. A freeze stops all behaviors in their tracks, which allows to analyze a snapshot of the dynamical system. This is useful to analyze a system at some point in time.

Fifth, a debug behavior runs in *zero execution time*. This means that when a debug behavior is activated, it freezes the system, then executes the body while everything remains frozen, and the system is defrosted when the debug behavior completes. This allows the debug behavior to run checks and/or to update logs accordingly at specific points in the runtime. The zero execution time is not useful when studying certain racing conditions. When a debug behavior has no body, it doesn't need to execute and it doesn't freeze the system. This assumes that their activation is sufficient as debug information.

Handling exceptions

So far, we haven't considered the case where an illegal operation is executed at runtime. Such actions include the use of a null pointer, division by zero, and many other issues. These problems can occur both in the triggering condition evaluation as when executing the body.

It is undesirable that the system crashes on such events. Multi-Agent Systems make it possible to build more robust systems, and it would be too bad if the entire system crashed on a single runtime bug.

In traditional languages, an error event is thrown that is caught at the end of the behavior. If the event is not caught, then the event is propagated outward until ultimately the entire system may crash.

According to the present invention, a similar system is used that exploits the event model. We will define a system pre-defined scalar property "exception" associated with a behavior. Like the subsumed property, the exception property is a true sensor. When it is changed, it will throw events.

As a result, the designer can use a completeWhen statement to catch exceptions as they occur, and handle them.

If a behavior doesn't handle its exception, the behavior completes immediately. The exception is not propagated in any way, and the system continues to work.

Since the exception is a true sensor, any other behavior in the agent can respond to it. Therefore, another behavior in the agent may be dedicated to handling exceptions that occur in the agent.

Exhaustive behaviors: find self triggering

The compiler will try to detect self-triggering of exhaustive behaviors. In particular, it will look at the agents in triggering condition of the exhaustive behavior, including any operators, and will look at the sensors updated in the body of the exhaustive behavior. If the system identifies a match, then a compiler severe warning will be generated. A risk of self-triggering exists.

In the dependency tree, if a loop can be found that starts with an exhaustive behavior, and that ends with the same exhaustive behavior, and that has only exhaustive behaviors in its path, then a severe warning will be generated by the compiler. In this case, a circular self-reference exists that will likely lead to taking up all system resources.

Matrix Analyzer

The Matrix Analyzer is a tool to monitor agents as they are executing. Like an analyzer in electronics, it continuously shows the values of all relevant parameters. In the matrix Analyzer, there are various views.

One view shows the list of communities found on the computer that are in debug mode (otherwise the matrix analyzer cannot see them).

Another view shows a single agent, with all of its sensors and behaviors. For every sensor, the value is shown. For behaviors, the status of the code is show in colors, e.g.:

- Black means the code is not currently active.
- Green means the code is executing.
- Orange means the code is waiting for execution.
- Red means something went wrong with the code during execution.

A third view represents every agent as a single block or dot. The color shows the status of the agent, e.g.:

- If something went wrong with any part of the code of the agent during execution, the agent is red;
- If the agent isn't red and any behavior is waiting for execution, the agent is orange;
- If the agent isn't red or orange, and any behavior is active, the agent is green.
- Otherwise, the agent is black.

The third view can be represented as a matrix. As a result, a screen of 1600 x 1200 pixels can show the activity of up to 1.92 million agents.

In the third view, a particular pixel can be indicated, which will bring up the second view for this agent. This allows to analyze a colored dot in more detail.

IDE LEVEL ASPECTS

Agent view

To visually monitor agents, a visual agent designer is defined. There a a number of views in such an visual modeler.

In one view, the developer sees the agent in a way similar to UML. Instead of the usual single indicator before the variable or method in object oriented (OO) modeling, we use two indicators. The first is the same as with OO. It indicates if the method, behavior, variable or sensor is private, public or friend. The second indicator shows where it is an method or a behavior, and if it is a variable or a sensor. The second indicate essentially indicates if it is an agent concept (sensor or behavior) or an object concept (variable or method).

While this indicator may seem quite simple, it is a true innovation. Indeed, it is the direct result of the definitions made elsewhere in this text. Because agents are defined in such a similar way to objects, this single indicator become possible. Because of the simplicity of the notation, the way it is shown is very similar to concepts the developer already knows. It is therefore very easy to use, and much easier to use that alternative proposals.

Behavior view

In the behavior view, the sensors and behaviors are shown outside of the agent they belong to. They are annotated by the name of the agent, but sensors and behaviors of the same agent do not need to be shown in the same neighborhood. Rather, the behaviors and sensors are spread out according to dependency, with the leave sensors and behaviors shown at the bottom (or at the top, or from left to right, or from right to left, according to the preferences of the user).

The sensors and behaviors are connected together with arrows, that show the dependency of between the items.

Dependency view

Based on the concepts of RIDL, a graph of dependencies can be defined. This graph can be visualized in at least two ways.

Either the agents can be shown to the user (as in the agent view), and arrows drawn between the agents to show the dependencies. Notice that the exact priority may vary at runtime and cannot be shown statically.

Alternatively, the agents can be shown in behavior view, and again arrows drawn between the behaviors and sensors to show the dependencies.

Community view

See debug level aspects for the community view. The community view shows the agents at runtime.

LEARNING AGENTS: NEURAL AGENTS - ASPECTS

In artificial neural networks (ANNs), neurons are mathematical formulas. They provide a number that represents their triggering value. Thresholds are often used to create only 0 or 1 as a triggering value. The formula in the neuron is based on a value often calculated as the sum of all nodes at a lower level, where each node is multiplied by a dedicated multiplication value. In the case of discrete ANNs, the weight must be between 0 and 1. By changing the individual weights, and by putting the neurons on layers, the system can learn to process complex data (such as identifying objects on images) simply through teaching. There are many books that explain in great details the concepts of Artificial Neural Networks.

In RIDL, the concept of a neural network is used with slight modifications. A neuron is represented by an agent. Its triggering value is a sensor. It has a behavior that responds to events from triggering values in a lower layer, to recalculate its triggering value.

The net result is that using the concepts of ANNs, RIDL software can learn very complex tasks without programming the solution (hence through training). As with ANNs, configuration of such a neural agent network is more an art than a science. However, the results can be surprising in complexity and quality.

To create layers of agents, various principles can be used. Inheritance can be used to put all agents of a single layer under a single class name. Or every agent can have a number of the layer it is in, and this number can be one of the conditions checked in

In one aspect, there is no "for all" construct in RIDL. It is impossible to visit all agents, because agents can be created or destroyed in the middle of such an action. Therefore, it is up to the agent itself to keep a list of agents it is connected to. Such a list *can* be traversed. In fact, the list can be kept up to date by using any-operators (if any agent exists with in the layer I'm monitoring, for which hold that it is not in my list, then add it to the list).

SELF-WRITING LEARNING AGENTS: GENETIC EVOLUTION OF SOFTWARE - ASPECTS

Introduction

Simply put, genetic programming works on two principles:

- mutations make small random changes
- cross-over takes to parents, and creates a single child by taking some parts of one parent, and some parts of the other parent.

Based on various application dependent parameters, the success factor of an agent can be determined. There are many systems to relate the overall success of the community to the success of the individual agent. In our human society, money is a major contributor to this success attribution.

Most successful agents are allowed to "breed", and using the two principles above, offspring are created. Variations exist, where a number of parents move to the next generation without change if they are extremely successful. The new generation is again measured for its success, and the new generation can again breed to create yet a newer generation. Every generation, the previous generation dies, although life expectancy of an agent may sometimes be multiple generations. Human life expectancy in the western world is today 4 generations.

As seen, genetic programming provides a way to allow software to evolve automatically to more efficient software.

Source Code Evoluti n

In the past, genetic programming has been successfully applied to a number of domains. In every genetic application, one of the main questions is what are the genes. What will be used as the basic material that will form the "DNA" of an agent?

Successful applications have found application specific "genes" that are basic to the convergence of the system to a solution.

In the past, there have also been experiments to use the keywords of a software language as the genes. The definition of the DNA is then the entire written software, and the phenotype (expression of the genes in practice, i.e. the way we look and act) is the result of the software.

These approaches to source code evolution don't work very well, because random mutation and blind cross-over have a very high probability of introducing bugs into the program. As a result, the further the software evolves, the worse it gets, until it entirely crashes. In jargon, this means that the system does not converge to a local optimum, and certainly not to a global optimum.

Triggering Conditions as basis for DNA mutations

Behavior-Based Multi-Agent Systems, as we have defined them in this document, are a natural match for genetic programming. It provides a very clear distinction between agents, and within agents behaviors form discrete blocks of functionality. This provides information which can guide the mutation and cross-over operators to be more efficient than in blind source code modification. In fact, in our system, the system never creates buggy code, and the system ultimately will converge to a global optimum given the success factor used.

Genetic programming assumes that we have a large number of agent classes, and only one instance of each agent class. Hence, every agent is an individual and unique.

When applying mutation or cross-over, unless otherwise stated, we take the body of one of the parents. It is important that the initial population of agents either contains behaviors with bodies that contain *learning* code (e.g. neural agents), or contains a lot of behaviors that take all sorts of small actions.

Let's first look at the mutation operator. The mutation operator works on one behavior of one agent. In one aspect, a number of mutation operators are defined, which work randomly over the population with an application-specific frequency:

- The name of a sensor that occurs in the behavior (usually in the triggering condition) is replaced by another existing name of a sensor. This replacement is done consistently, hence all occurrences are replaced to keep the logic of the software intact. Sensors can only be replaced by sensors of the same type.
- The name of a behavior that occurs in the behavior in the triggering condition is replaced by another existing name of a behavior.
- If a sensor is mentioned in the when part of the triggering condition, wait for a different event of the same sensor. Hence, an sensor.updates can be changed in either sensor.changes or sensor.event.

- If a behavior is mentioned in the when part of the triggering condition, wait for a different event of the same behavior. Hence, an behavior.activates can be changed in either behavior.completes, or behavior.event, amongst others.
- An event of the when part of the triggering condition can be dropped.
- A condition of the if part of the triggering condition can be dropped.
- An additional condition on any sensor already in the when part of the triggering condition can be added.
- An additional event of an existing behavior or sensor can be added to the when part of the triggering condition.
- A new sensor can be created in the agent, and the sensor is added to the when condition of the behavior. The updates to this sensor can come from the first specified mutation. The sensor is public or private depending on some probability.

Agent-level cross-over

A new agent can be constructed from two agents, by taking a number of behaviors from one agent, and a number of behaviors from the other agent. These behaviors are brought together into a new agent. All local sensors are of both behaviors are copied to the new agent, except for the local sensors that are not used in any of the behaviors.

Basic behavior-level cross-over

This cross-over works with two behaviors. A new behavior can be created by merging partial copies of the triggering conditions of both behaviors into a new triggering condition.

The body of the new behavior is taken from one of both parents. The code inside the body is not touched, leaving the algorithms intact.

If the parent from which we copy has a completeWhen clause, then copy that clause also identical. This ensures that the error handling associated with the algorithm is retained. Mutation on sensor names also applies to completeWhen clauses.

Sequential behavior-level cross-over

Another cross-over operator can make the two parents sequential. This cross-over operator takes one parent, and at the end of the body of that parent, it puts a continueWhen statement with the triggering condition of the second parent. After that, it adds the body of the second parent. All completeWhen clauses of both parents are then appended.

Sequential behavior-level mutation

When a behavior has continueWhen statements, the code starting from the start of the body, or starting from a continueWhen statement, until the next continueWhen statement, or until the end of the body, is deleted.

Other operators can be used. The key point is that the triggering conditions and the syntax of RIDL allow an algorithm to define clear points where it can safely paste code together, without breaking the software from a syntactical and semantical level.

Extended c mmunities in acti n

Genetic programming of multi-agent systems is made possible using the concept of similar communities.

A genetic program has access to its own source code, because the developer supplied a representation of the source code to the genetic program. The genetic program makes changes to the source code, and recompiles the code. While doing so, it is useful for the genetic program to make use of inheritance.

After compilation, the new program is started and as a result, a new community is created. This community is normally similar to the original community. Because inheritance was used, the agent classes of the old community are largely intact, but new offspring has been created.

Next, all agents are made to migrate to the new community. After this has happened, the old community is destroyed. The net result is that our agents are still the same, but are now in an environment where they need to compete with their offspring.

AGENT FILE SYSTEM ASPECTS

When a file system is based on a database, as is the case in Microsoft Windows Longhorn, then the agent-oriented database principles can be used to assign behaviors to files. For instance, a file may monitor itself and decide it needs to backup itself, or repair itself, or notify the user of some condition, or be in other ways self-managing. This would advantageously take the burden of PC maintenance away from the user.

ADDENDA

Appendix A:

- RIDL 2004 Programmer's Manual includingvarious examples written in the syntax of RIDL 2004

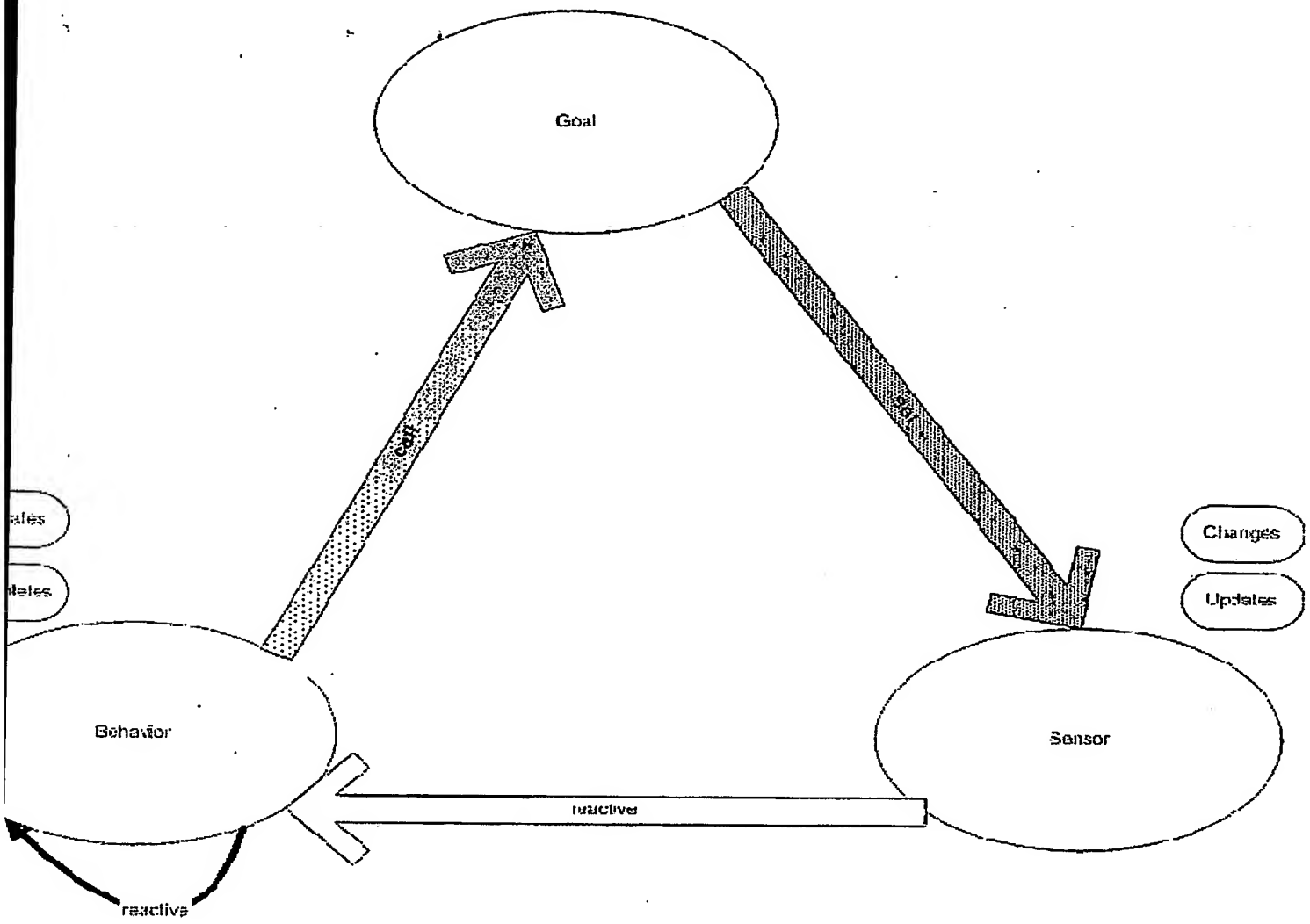

FIG. 1